Resilient file uploading with Go

Marius Kleidl

FOSDEM, Go devroom
1 February 2026

Hello, I'm Marius Kleidl 👋

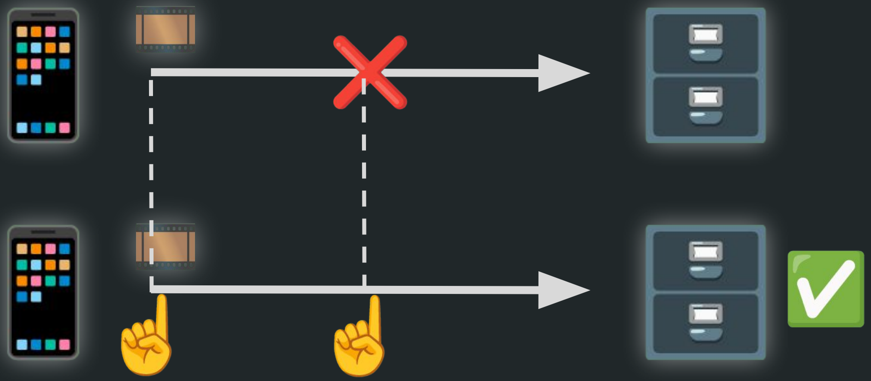
Senior Backend Engineer @ GetStream.io

🌐 mariuskleidl.net

📧 hello@mariuskleidl.net

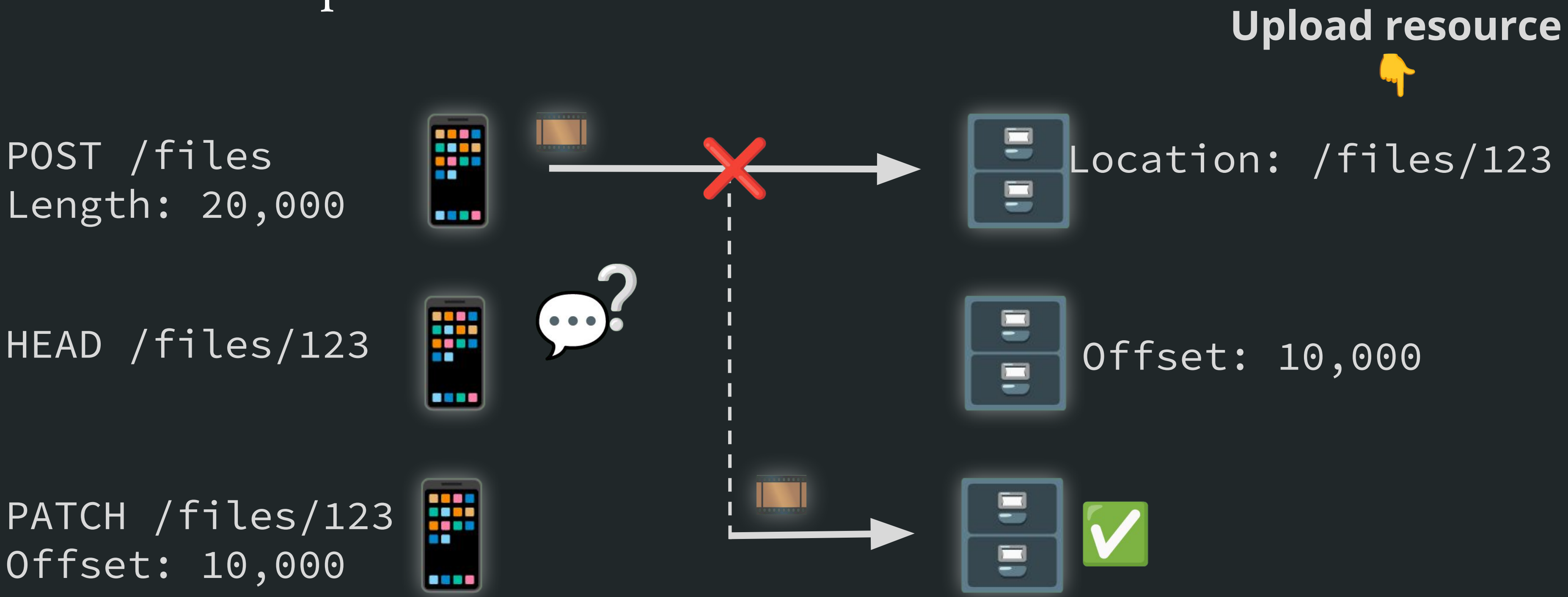
🟦 linkedin.com/in/marius-kleidl

Typical file uploads over HTTP



👉 File data has to be re-transmitted again 😥

Resumable uploads to the rescue!



Tus.io: resumable uploads for everyone

Open, HTTP-based protocol for resumable uploads

Open source libraries that are easy to integrate everywhere

A naive implementation

```go
func handleUpload(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    upload, err := state.GetOrCreateUpload(ctx,
                        r.PathValue("id"))
    if err != nil { /* ... */ }

    err = upload.Append(ctx, r.Body)
    if err != nil { /* ... */ }

    w.WriteHeader(http.StatusOK)
}
```

👈 Create new resource or get existing one

👈 Append request body to upload

A naive implementation

```go
func (u *S3Upload) Append(ctx context.Context, reader io.Reader) error {
    f, err := os.CreateTemp("", "upload-part")
    if err != nil { return err }
    defer f.Close()

    _, err = io.Copy(f, r.Body)
    if err != nil { return err }

    err = u.S3.UploadAsPart(ctx, f)
    if err != nil { return err }

    return nil
}
```

👉 Buffer body in temporary file

‼️ Returns error when read fails,
e.g. i/o timeout, unexpected EOF

👉 Transfer data to S3

‼️ No data is stored on S3

```
type happyReader struct {
    src io.Reader
    err error
}

func (r *happyReader) Read(p []byte) (int, error) {
    if r.err != nil { return 0, io.EOF }
    n, err := r.src.Read(p)
    if err != nil {
        r.err = err
        err = io.EOF   👉 Mask an error as io.EOF
    }
    return n, err
}

func (r *happyReader) Err() error {
    return r.err   👉 Allow for later retrieval
}
```

```go
ctx := r.Context()
```

‼ Context gets cancelled if request is cancelled

```go
upload, err := state.GetOrCreateUpload(ctx, r.PathValue("id"))
if err != nil { /* ... */ }

reader := &happyReader{src: r.Body}
```

👉 Wrap request body

```go
err = upload.Append(ctx, reader)
if err != nil { /* ... */ }
```

‼ No upload saved if context is cancelled

👉 Is shielded from read context and upload chunk is saved with 🙌 error shielding

```go
err = reader.Err()
```

👉 Check read errors afterwards

```go
if err != nil { /* ... */ }
```

```go
func newDelayedContext(parent context.Context, delay time.Duration)
context.Context {
    ctx, cancel := context.WithCancel(context.Background())

    go func() {
        <-parent.Done()
        <-time.After(delay)
        cancel()
    }()

    return ctx
}
```

👈 Propagate cancellation after delay

```go
ctx := newDelayedContext(r.Context(), 10*time.Second)
```

👆 Delay cancellation by ten seconds

```go
upload, err := state.GetOrCreateUpload(ctx, r.PathValue("id"))
if err != nil { /* ... */ }

reader := &happyReader{src: r.Body}

err = upload.Append(ctx, reader)
if err != nil { /* ... */ }
```

👉 Saving upload chunk has now additional time 🙌

```go
err = reader.Err()
if err != nil { /* ... */ }
```

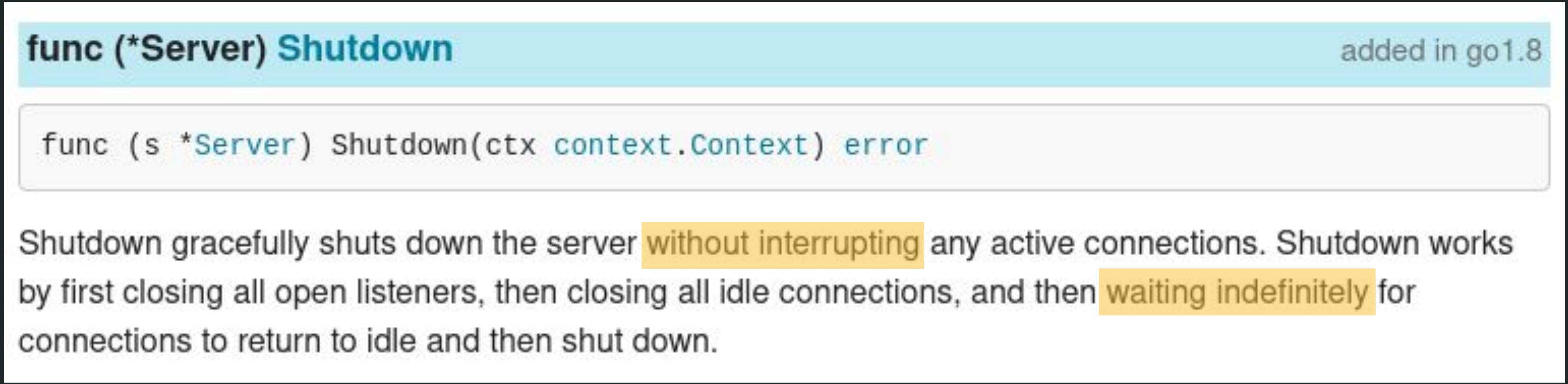
```go
func TestNewDelayedContext(t *testing.T) {
    synctest.Test(t, func(t *testing.T) {
        parent, cancelParent := context.WithCancel(t.Context())
        delayedCtx := newDelayedContext(parent, 5*time.Second)
        cancelParent()

        time.Sleep(5*time.Second - time.Millisecond)
        synctest.Wait()
        assert.NoError(t, delayedCtx.Err())

        time.Sleep(time.Millisecond)
        synctest.Wait()
        assert.Equal(t, context.Canceled, delayedCtx.Err())
    })
}
```

👈 Cancel parent context

👈 Should not be cancelled yet

👆 Should now be cancelled

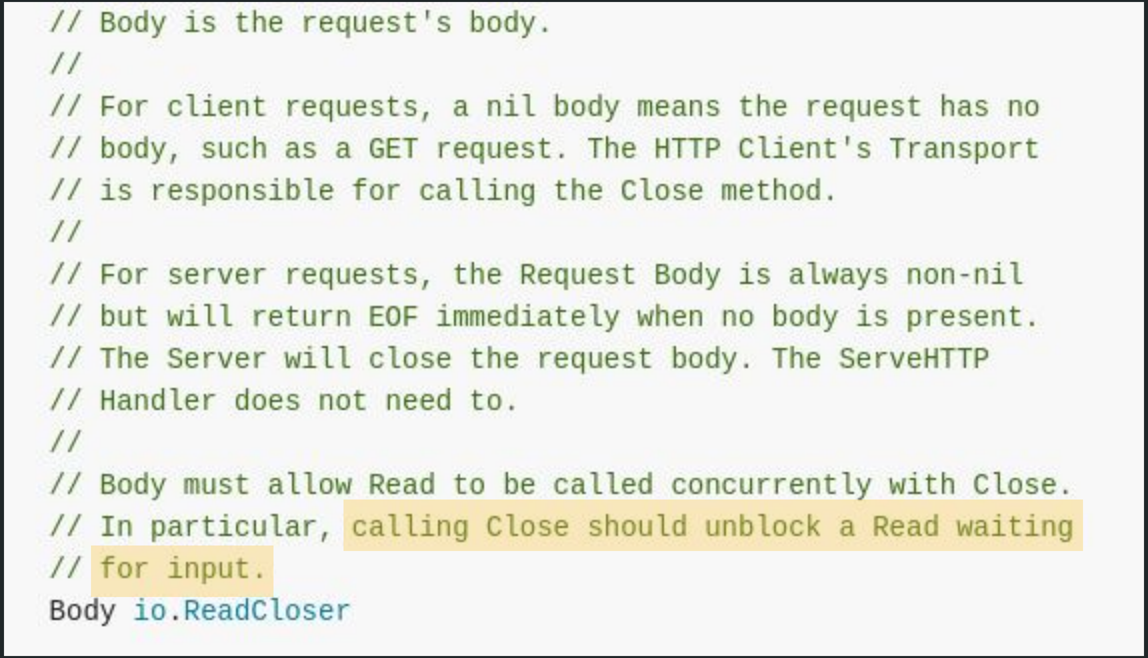
Shutting down servers


func (*Server) Shutdown added in go1.8

```
func (s *Server) Shutdown(ctx context.Context) error
```

Shutdown gracefully shuts down the server without interrupting any active connections. Shutdown works by first closing all open listeners, then closing all idle connections, and then waiting indefinitely for connections to return to idle and then shut down.

⚠️ Long running requests are not stopped
⚠️ Cancelling context is not enough because `io.Copy` can hang
⚠️ There is no `io.CopyContext`



```go
// Body is the request's body.
//
// For client requests, a nil body means the request has no
// body, such as a GET request. The HTTP Client's Transport
// is responsible for calling the Close method.
//
// For server requests, the Request Body is always non-nil
// but will return EOF immediately when no body is present.
// The Server will close the request body. The ServeHTTP
// Handler does not need to.
//
// Body must allow Read to be called concurrently with Close.
// In particular, calling Close should unblock a Read waiting
// for input.
Body io.ReadCloser
```

```go
serverCtx, cancelServerCtx := context.WithCancel(context.Background())
```

👆 Server's context

```go
ctx, cancel := context.WithCancel(r.Context())
ctx = newDelayedContext(ctx, 10*time.Second)

go func() {
    select {
    case <-serverCtx.Done():
        r.Body.Close()
        cancel()
    case <-r.Context().Done():
        // Nothing to do
    }
}
```

👈 Interrupt reads on shutdown

👉 Uploaded data is saved on shutdown 🎉

Key takeaways

1️⃣ Long-running requests can be tricky but…

2️⃣ Handle errors gracefully through shielding

3️⃣ Use contexts creatively

4️⃣ `net/http` has hidden gems everywhere

5️⃣ Resumable uploads implementations: tus.io

🌐 mariuskleidl.net

📧 hello@mariuskleidl.net

🟦 linkedin.com/in/marius-kleidl